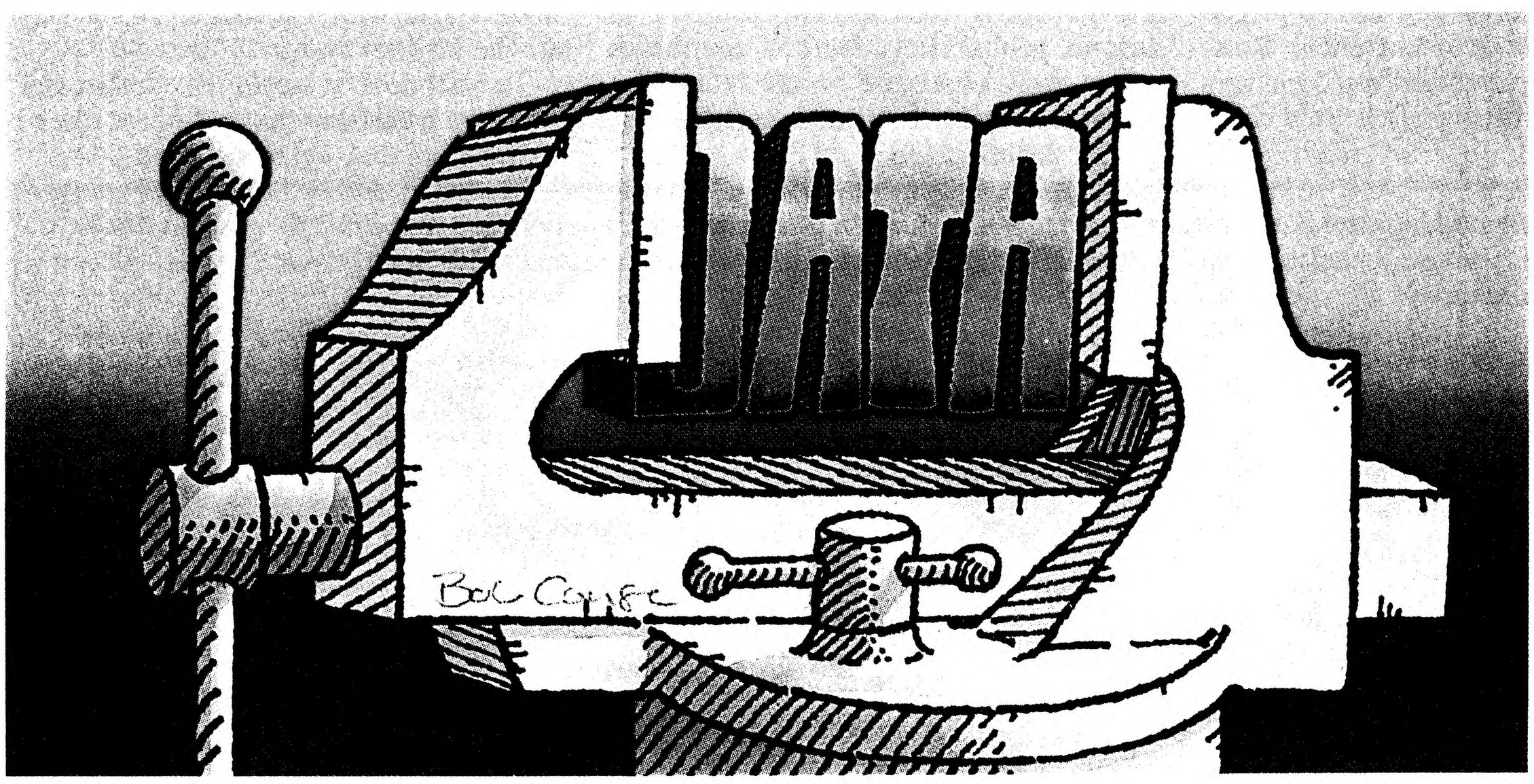
STEVE APIKI

LOSSLESS DATA COMPRESSION



At first glance, the concept of data compression seems too good to be true. The idea of shrinking information without losing any of it looks to be a something-for-nothing proposition that violates what should be one of Newton's lesser-known laws: the law of conservation of data.

Despite the aura of mystique that surrounds it, data compression is based on a simple idea: mapping the representation of data from one group of symbols to another, more concise series of symbols. Data-compression programs and dedicated compression hardware use several different algorithms to achieve this end.

Two compression schemes, Huffman coding and LZW coding (for Lempel and Ziv, its creators, and Welch, who made substantial modifications), form the basis for much of the compression that we use from day to day. These techniques also represent two distinct schools of compression algorithms. An understanding of how each algorithm works provides an excellent background in compression in general.

Both Huffman and LZW coding are *lossless* compression techniques. They are appropriate to use for compressing any kind of data because the expanded representation is identical to the original input to the compressor. Joint Photographics Experts Group (JPEG), Motion Picture Experts Group (MPEG) (see "Putting the Squeeze on Graphics," December 1990 BYTE), and other cutting-edge image-compression algorithms achieve fantastic compression ratios at the expense of exact data reproduction. These techniques work well for images and sound data, but they are not appropriate for general data.

Two algorithms—Huffman coding and LZW coding—are at the root of most compression

Huffman coding, originally proposed sometime in the early 1950s, reduces the number of bits used to represent frequent characters and increases the number of bits used for infrequent characters. The LZW method, on the other hand, encodes strings of characters, using the input stream to build an expanded alphabet based on the strings that it sees. These two very different approaches both work by reducing redundant information in the input data.

Huffman Coding

Huffman coding is probably the best-known method of data compression. The simplicity and elegance of the technique have made it a longtime academic favorite. But Huffman codes also have practical applications; for example, static Huffman codes are used as the last stage of JPEG compression. The MNP-5 data-compression standard for modems (see "4800 Bits, No Errors," June 1989 BYTE) uses dynamic Huffman compression as part of its process. Finally, Shannon-Fano coding, a close relative of Huffman coding, is used as one stage in PKZIP's powerful "imploding" algorithm.

continued

ILLUSTRATION: BOB CONGE © 1991

Huffman coding works on the premise that some symbols are used more often than others in data representation. The most common representation, the ASCII alphabet, uses 8 bits for each character. In English, the letter *e* is considerably more likely to appear than the letter *q*, yet we use the same number of bits to represent each. If we used only 4 bits for an *e* and 12 bits for each *q*, we would save some bits whenever storing English text.

Huffman coding formalizes this idea of relating symbol length to the probability of a symbol's occurrence. Static Huffman coding requires you to have a table of probabilities before you begin compressing the data. This table can be compiled from statistical observations (such tables have been compiled for inputs like English), or the compressor can prescan the input data to find the symbol probabilities before it starts to compress the data.

The compressor and decompressor can construct an encoding tree with this probability information. The encoding tree is a binary tree with one leaf for each symbol. To construct the tree, the compressor starts with the two symbols of lowest probability. It then combines these two as two leaf branches under a node; this node, in turn, is assigned the sum of the two probabilities. The compressor then considers this node along with the rest of the symbols in the probability list, and it again selects the two least probable items. It continues to build and combine nodes until it builds a single tree, with the probability at the root equal to 1.

The resulting tree has leaves of varying distance from the root. The leaves that represent the symbols with the highest probability are closest to the root, while those with the lowest probability are the farthest away.

To encode a symbol, the compressor finds the path from the root of the tree to

Listing 1: *Dynamic Huffman compression/expansion pseudocode. All structure references are simplified for readability. Unless explicitly noted, structures are elements of the* `Tree` *array. For example,* `char.parent` *should properly read* `Tree[char].parent`.

```
PROCEDURE huffman compress
tree <- ROOT                              // initialize the tree
add_node (empty leaf, ROOT, 0) // add the empty leaf to the tree
char <- (next character from buffer)// read in the first character
add_node(char, ROOT, 1)                   // add this char to the tree
write char to output buffer               // send the first character
                                          // as a literal character
WHILE (input buffer not empty)
                                          // read in a character
   char <- (next character from buffer)
   IF (char is not known)
      transmit(empty leaf code)
      write char to output buffer  // send the literal character
      update_tree (empty leaf code)// adjust the tree
      IF (all nodes not full)
                                          // add to the tree
         add_node(char, empty leaf, 1)
                                          // move empty leaf
         add_node(empty leaf, empty leaf, 0)
      ELSE                                // last node to add
                                        // assign empty leaf info to char
         char.parent <- empty leaf.parent
   ELSE                                   // this character is known
      transmit(char code)
      update_tree (char code)             // adjust the tree
CONTINUE

PROCEDURE huffman expand
tree <- ROOT                              // initialize the tree
add_node (empty leaf, ROOT, 0)  // add the empty leaf to the tree
char <- (next char from buffer) // read in literal first character
add_node(char, ROOT, 1)                   // add this char to the tree
write char to output buffer               // write the first character
WHILE (input buffer not empty)
   char <- incode(buffer)                 // read in a code
   update_tree (char code)                // adjust the tree
   IF (char=empty leaf)
      char <- next char from buffer// read in literal character
      IF (all nodes not full)
                                          // add to the tree
         add_node(char, empty leaf, 1)
                                          // move empty leaf
         add_node(empty leaf, empty leaf, 0)
      ELSE                                // last node to add
                                        // assign empty leaf info to char
         char.parent <- (empty leaf.parent)
   write char to output buffer
CONTINUE
```

```
// Huffman compression support routines
// assume expander has same tree structure as compressor for
// readability. In reality, each expansion tree node has daughter
// pointers as well as a parent pointer.

                                          // add a node to the tree
PROCEDURE add_node (code, parent, branch)
code.parent <- parent                     // assign parent pointer
code.bit <- branch                        // assign bit for this code
code.weight_ptr <- (next available spot in weight list)
IF (code is a character)
  Weightlist[code.weight_ptr] <- 1
ELSE                                      //code is the empty leaf
   Weightlist[code.weight_ptr] <- 0
ENDIF

PROCEDURE update_tree(code)
WHILE (code != ROOT)
  Weightlist[code.weight_ptr] ++    // increment weight
  IF (Weightlist[code.weight_ptr] = MAXWEIGHT)
       scale(weightlist)
   IF (Weightlist[code.weight_ptr] > Weightlist[code.weight_ptr-1]
                                          // if weight is greater than
                                          // that of node listed above
                                          // it in the weight list
      swap_node<-(heaviest node, which is lighter than code)
      IF (swap_node != code.parent)// don't swap if parent-child
         swap(swap_node, code)
   code<-code.parent
CONTINUE

PROCEDURE transmit (code)          // transmit a Huffman code
DO
   push code.bit                  // push this code's bit on a stack
   code<-code.parent                 // move to parent node
WHILE (code != ROOT)
pop bitstack                      // send the bits to output

PROCEDURE incode (buffer)         // read a Huffman code
code<-ROOT                         // start at the root
DO
   bit<-(next bit from buffer)     // read in a bit
   IF (bit=0)
      code<-code.zero_daughter     // jump to zero child
   ELSE
      code<-code.one_daughter      // or one child
WHILE (code.daughter != NULL)      // until you find a leaf
RETURN code                       // leaf location is its value
```

the symbol's leaf. Suppose the compressor wants to encode the letter *s*. It starts at the leaf corresponding to *s* and jumps to the parent node, noting which branch (0 or 1) it was on. It continues to jump up the tree until it reaches the root. The list of branches, when reversed, describes the path from the root to *s*: This is the symbol's Huffman code.

High-probability characters are close to the root, so their codes are short. Low-probability characters are far from the root and have longer codes.

To decode, the decompressor takes the code and processes it in reverse. That is, it starts at the root of the tree. If the first bit in the code is a 1, it jumps to the node on the 1-branch from the root. It continues reading bits and jumping until it reaches a leaf; the symbol at the leaf is the decoded character.

One more property of the Huffman tree bears discussion. Because symbols are always leaves, symbol nodes never have any children. When the decompressor gets to a leaf node, it knows to stop reading from the input immediately because it knows it has reached a leaf. In other words, one Huffman code is never the prefix of another. This means that although code lengths are variable, the compressor always knows when one code ends and another begins, and there is no need to explicitly place delimiters between codes.

Dynamic Huffman Coding

The greatest difficulty with Huffman codes, as you probably noticed from the discussion above, is that they require a table of probabilities for each type of data to compress. This is not a problem if you know you will always compress English text; you simply provide a suitable English text tree to the compressor and decompressor. The JPEG protocol defines a default Huffman tree for compressing JPEG data. In the general case, when you don't know the symbol probabilities for your input data, static Huffman codes can't be used effectively.

Fortunately, a dynamic version of Huffman compression can construct the Huffman tree on the fly while reading and actively compressing. The tree is constantly updated to reflect the changing probabilities of the input data.

Listing 1 contains a pseudocode version of a dynamic Huffman compression/decompression program. The actual code, which is available from the usual sources, is written in 8088 assembly language. These programs are based on an algorithm described in reference 1, which cites a number of original sources. Reference 2 presents a more efficient, although complex, algorithm for dynamic Huffman compression.

The key to starting with an uninitialized tree is the introduction of an empty leaf. The empty leaf is simply a leaf node with no symbol attached to it; this leaf has zero probability. The initial tree, held by both the compressor and decompressor, has only the root and a single empty leaf.

The compressor starts the ball rolling by reading in a character. It attaches this character to the 1-branch of the root, leaving the empty leaf on branch 0. It then sends this character to the decompressor as a literal ASCII code, and the decompressor makes the same adjustment to its tree.

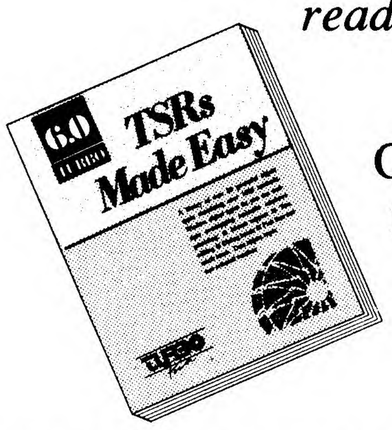
For each character read thereafter, the compressor performs the following steps. First, it checks to see if the code is

"Writing a TSR is exceptionally easy" ... and now it's inexpensive too!

Now you can turn Turbo Pascal programs into rock solid TSRs with ease. **TSRs Made Easy** lets you create conventional TSRs or swapping TSRs that use only 6K of RAM. **TSRs Made Easy** provides ■ TSR swapping to EMS, XMS, or disk ■ selectable hot keys ■ keyboard macros ■ unloadable TSRs ■ 8087 TSR support ■ interface to transient programs ■ ISR handling, and more.

TSRs Made Easy includes full source, complete documentation, and plenty of small example and demo programs. You pay no royalties.

"Writing a TSR. . .is exceptionally easy. The documentation is extremely readable and well done."
Computer Language, May 1990

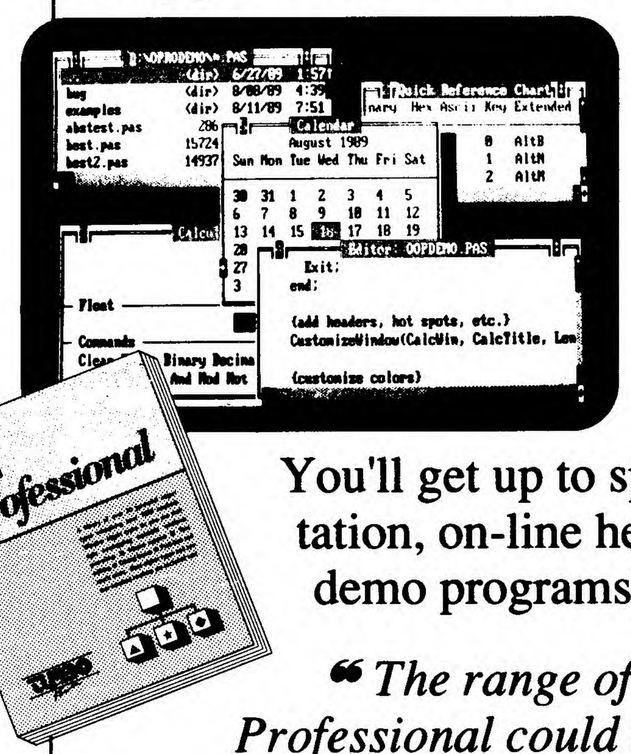


One of programming's most formidable tasks is now very simple. . . and very affordable!

TSRs Made Easy, only $49.

New! Object Professional for Turbo Pascal 6.0

Object Professional version 1.1 is fully updated for Turbo Pascal 6.0. New are SAA/CUA style dialog boxes, draggable windows, XMS/EMS 4 support, and more.



Object Professional includes over 100 object types that will multiply your productivity. Included are windowing and menu systems, menu and data entry screen generators, data object types, and routines that provide swapping TSRs. You'll get up to speed fast with clear documentation, on-line help, full source code, and hot demo programs.

"The range of objects is fantastic. Object Professional could save you man-years of effort."
Jeff Duntemann

Object Professional, only $189.

Call toll-free to order: 1-800-333-4160

TSRs Made Easy has exactly the same TSR routines as OPro. TSRs Made Easy requires Turbo Pascal 6.0, 5.5 or 5.0. OPro requires Turbo 6.0 or 5.5. 5.25" and 3.5" disks included. Add $5 per order for standard shipping in U.S./Canada. Call for other shipping charges. Registered owners of OPro may update to version 1.1 for $20, include your serial number.

TURBO Power

9AM–5PM PST Monday through Friday, USA & Canada.
For more information call (408) 438-8608, fax to (408) 438-8610,
or send mail to CompuServe ID 76004,2611
TurboPower Software PO Box 66747 Scotts Valley, CA 95067-0747

in the encoding tree. If the code is there, the compressor sends it in the same fashion as in the static case. If not, it sends the code for the empty leaf. Then it sends the new character as a literal ASCII code. Finally, the compressor adds two codes, one for a new empty leaf on branch 0 and one for the new code on branch 1. When the tree is full (i.e., when all characters have been seen), the compressor just changes the last empty leaf node into the last character.

The decompression program can make adjustments to its tree because it has exactly the same tree as the compressor. When it receives an empty leaf code, it reads the next code from the compressed data as an ASCII literal. It then employs the same update routine as the compressor uses to update the tree.

The empty leaf and the uninitialized tree don't solve the problem of keeping track of changing probabilities, however. To do that, you need to introduce weights to each node in the tree and update these weights as you process the input data. You also need to maintain a list of node designations (and weights) sorted by weight.

Each character starts at weight 1 (the empty leaf starts at 0). Whenever the compressor transmits a character that is in the table, it increments the weight of that character's node. If this change makes the character node heavier than nodes that are listed higher in the weight list, the compressor swaps the character node with the heaviest node that is lighter than the character node. By *swapping*, I mean trading parent nodes and branch designations only; the children of the swapped nodes are not affected, so there is no danger of a leaf node becoming internal, or an internal node becoming a leaf.

The compressor then jumps up the tree to the character's parent, which may have changed with the last swap. It continues the process with the parent and on up the tree until it gets to the root.

The figure shows the early stages of dynamic Huffman tree construction for a very simple input. You can follow the addition of new leaves via the empty leaf mechanism as well as by node swapping in this diagram.

Huffman Gotchas

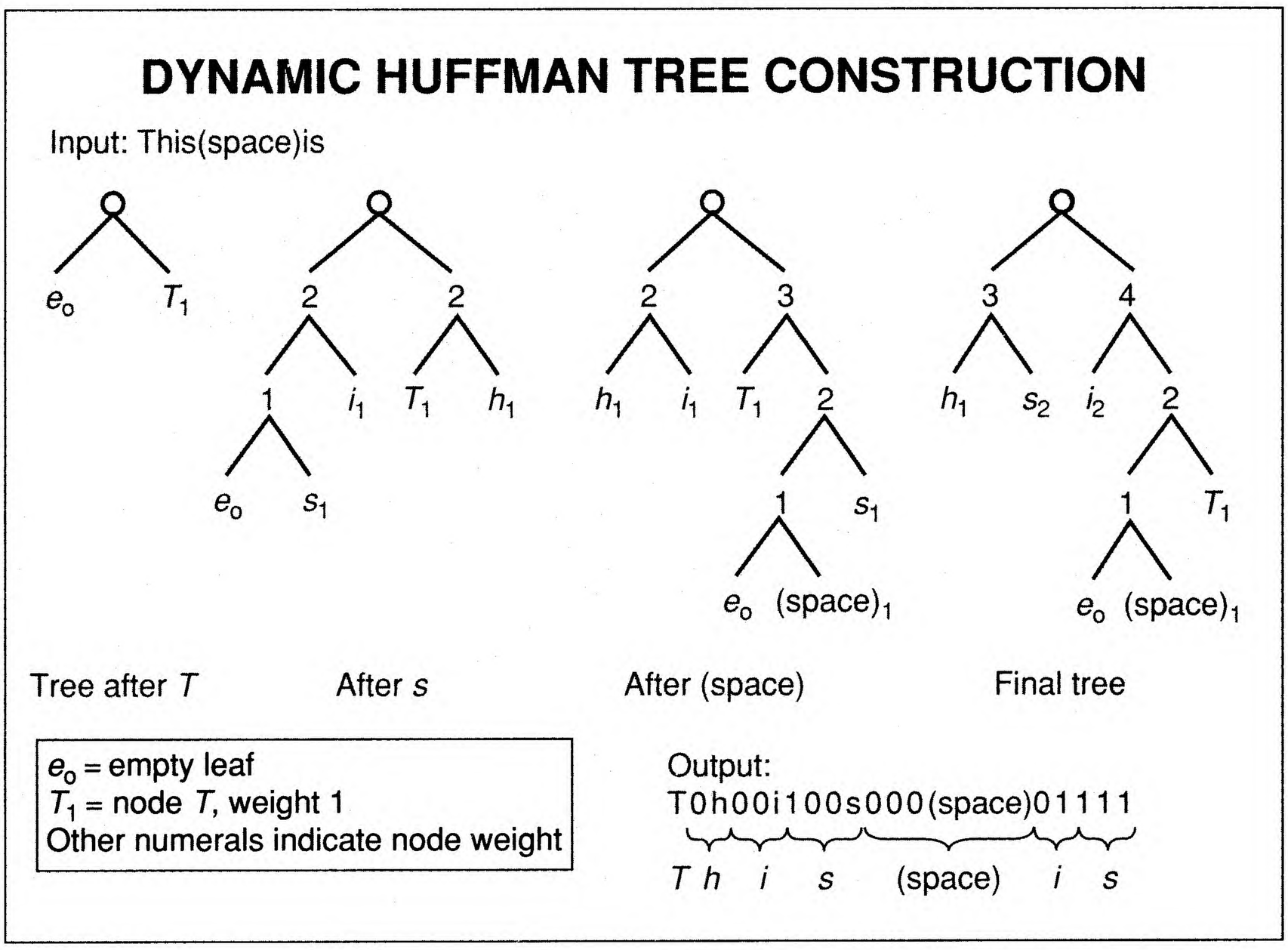
As usual, there are a few snags when you're actually implementing the dynamic algorithm, regardless of its elegance. The first problem is that you can't perform node swapping while transmitting a code, although both require you to start at a character node and hop up the tree parent by parent. You can't do the two procedures at the same time, because swapping nodes causes the parent to change, which causes the code transmitted to change. You would send a code to the decompressor before it knows what to do with it.

A way around this dilemma is to make two passes in the compressor—one for transmitting and one for updating. The decompressor also makes two passes—one for receiving (going down the tree) and one for updating (going back up).

The second problem occurs because of the empty leaf. Because the empty leaf has zero weight, it is possible for a sibling of the empty leaf to become heavier than its parent at the start of the update process. However, swapping between child and parent will scramble the tree, leaving the parent as its own child. Fortunately, simply aborting any swap between child and parent solves the problem.

Finally, there isn't any way for the

The Periscope Company, Inc.
1197 Peachtree St., Atlanta, GA 30361, USA • 404/875-8080
FAX 404/872-1973



The Huffman encoding tree changes to respond to changing character probabilities in this dynamic example. At first, all the transmissions are empty leaf code/literal character combinations. When i *and* s *are transmitted a second time, the compressor uses their codes instead of literals. As* s *is reused, it moves higher up the tree, shortening its corresponding code.*

LZW COMPRESSION

Table 1: *An instance of compression, at code 258. The compressor saves a code by transmitting 258 instead of* is, *the literal representation. Strings are stored in the LZW table as code-character combinations rather than full strings.*

Input	Compression table	Compressed string	Expansion table
T	—	—	—
h	256←*T*+*h*	*T*	—
i	257←*h*+*i*	*h*	256←*T*+*h*
s	258←*i*+*s*	*i*	257←*h*+*i*
(space)	259←*s*+(space)	*s*	258←*i*+*s*
i	260←(space)+*i*	(space)	259←*s*+(space)
s	—	—	—
(space)	261←258+(space)	258	260←(space)+*i*
a	262←(space)+*a*	(space)	261←258+(space)

decompressor to detect the end of transmission if the compressor must send out full bytes (as in a file-compression program). Suppose, for example, a transmission is 81 bits long. When the decompressor reads the first bit of the eleventh byte, it has no way of knowing that it's the last significant bit and that the remaining 7 are garbage. Therefore, the file-compression code must prepend a file length to the compressed data, making it a few bytes longer.

LZW Compression

The LZW algorithm, which was first presented by Welch in 1984 (see reference 3), has become a widely used technique during the last few years. CompuServe's GIF file format uses LZW compression, as do ARC, Unix's `compress`, Stuffit, and PKZIP. The algorithm itself is patented by Sperry.

Although straightforward in concept, the LZW algorithm can be a little difficult to implement on a real machine with real constraints. Despite some complexities, however, the technique is powerful and fast enough to make it popular.

LZW works by extending the alphabet—it uses the additional characters to represent strings of regular characters. To use LZW compression on 8-bit ASCII codes, you extend the alphabet by using 9-bit or larger codes. The additional 256 characters that the 9-bit code gives you are used to store strings of 8-bit codes, which are determined from strings in the input.

The compressor maintains a string table with strings and their corresponding codes. The string table corresponds to the extended alphabet. Initially, the compressor starts with a string table with only the 256 literal codes defined. If you're using 9-bit codes, the string table has an additional 256 empty entries; if you're using 10-bit codes, it has 768 empty entries, and so on.

The compression algorithm works like this: Start with a null string. Read in a character, and append it to the string. If the string is in the string table, continue reading and appending characters until you find a string that is not. Add this string to the string table. Write the code for the last known string that matched the output. Use the last character as the basis for the next string, and continue reading until you run out of input. That's really all there is to it.

Table 1 shows an example of LZW compression, using the same simple input in the figure. The compressor reads in the initial *T* and appends it to the null string. The string *T* is a literal character, so it is in the table. Next, the compressor reads an *h* and looks up *Th* in the string table, where it doesn't find it. It adds *Th* to the table at the next available position and sends out the last known string, *T*. It continues reading characters and adding strings until the input is exhausted.

This short and simple sample input shows only one instance of compression, when the code 258 is sent out instead of the string *is*. If I were using 9-bit codes, I would have sent eight 9-bit codes to represent *This is a* for 9 bytes either way and for break-even performance. Longer, more realistic inputs, of course, let you build a longer and more effective string table. The more repetitive that strings appear, the more you can compress.

Unfortunately, this simple compression algorithm eats memory like popcorn. Every time the compressor finds a new string, it adds it to the table. Each string that it adds is of variable length, which can lead to a storage nightmare.

Luckily, there is a simple way out. As you may have noticed, each new string is actually an old string plus a new character. Instead of storing strings explicitly, you can store them as code and appended character combinations. Table 1 shows this storage method. Code 261, for example, is stored as 258+(space) rather than "*is*(space)", which is the string that it represents.

continued on page 386

continued from page 314

Listing 2: *LZW compression/expansion pseudocode. All structure references are simplified for readability. Unless explicitly noted, structures are elements of the* `Table` *array. For example,* `tableoffset.char` *should properly read* `Table[tableoffset].char`.

```
PROCEDURE LZW Compress
next_code <- MAXCHAR+1
numbits <- MINBITS                    // bits to represent a char + 1
basecode <- (next char from buffer)// read in a character
WHILE (input buffer not empty)
   char <- (next char from buffer) // read in another character
   IF (lookup(char, basecode, location)=FOUND)
                                      // this combination in the table
      basecode<-location.code         // update the base code
CONTINUE                              // get another character

// found a code that is not in the table
   outcode (basecode, numbits)    // send the last complete code
   add_code(char, basecode, next_code, location)
                                      // add basecode+char to table
   IF (table full)
      clear(codelist)                 // clear some entries and put
                                      // their codes in the list of
                                      // available codes
   IF (table has been filled)
                                   // get another code from the list
      next_code <- next code from codelist
   ELSE
      next_code++                        // or just use the next code
      IF (log2 (next_code) > numbits)// increase the bit size?
         numbits++
   basecode <- char                   // start of next string
CONTINUE
outcode (basecode, numbits)

PROCEDURE LZW Expand
next_code <- MAXCHAR+1
numbits <- MINBITS                    // bits to represent a char + 1
code <- incode (buffer, numbits) // read a variable-length code
write code to output                  // write out this character
lastcode <- code                      // start with this character
WHILE (input buffer not empty)
   code <- incode (buffer, numbits)// read another character
   IF (code not in table)             // is this the special case?
//  this is the special case handler for codes not in the table
      outstring(lastcode)             // send out the last string again
      write lastchar to output        // and a duplicate first char
   ELSE                               // the normal case
      outstring(code)                 // send the string for this code
                                      // get the new last char
   lastchar <- (first char from output string)
                                      // add a new table entry
   add_code(lastchar, lastcode, next_code)
   lastcode <- code
   IF (table full)
      clear(codelist)                 // clear some entries and put
                                      // their codes in the list of
                                      // available codes
   IF (table has been filled)
                                   // get another code from the list
      next_code <- (next code from codelist)
   ELSE
      next_code++                        // or just use the next code
      IF (log2 (next_code) > numbits) // increase the bit size?
         numbits++
CONTINUE
```

```
// LZW compression support routines

// Note: expansion string table is indexed, while compression table
// is hashed by char and basecode. Therefore, add_code and clear
// shown here are appropriate for the compressor.
// Actual add_code and clear used by expander are not
// as complex.

PROCEDURE lookup (char, basecode, tableoffset)
                                      // find the table entry (table
                                      // offset is passed by reference)
tableoffset <- hashfunction (char, basecode)
DO FOREVER
   IF (tableoffset is a filled table entry)
      IF( tableoffset.char = char AND
tableoffset.basecode=basecode)
         RETURN FOUND
      ELSE
         tableoffset <- rehash (char, basecode)
   ELSE
      RETURN NOT_FOUND
CONTINUE

PROCEDURE add_code (char, basecode, code, tableoffset)
tableoffset.code <- code           // update the fields at
tableoffset.char <- char           // location in the table
tableoffset.basecode <- basecode
x_index[code] <- tableoffset  // update the cross-reference table

PROCEDURE clear (codelist)           // clear part of the table
STATIC bits_in_oldest <- MINBITS  // keep track of oldest leaves
FOR (entry<-0) TO (entry=TABLESIZE)
   mark x_index[entry] as a leaf // mark every cross-index entry
FOR (entry<-0) TO (entry=TABLESIZE)
   notleaf <- x_index[entry].basecode
   unmark x_index[notleaf]           // unmark those used as other
                                     // node's basecodes
FOR (each entry represented by bits_in_oldest bits)
   IF x_index[entry] is marked as a leaf
                                     // erase the oldest leaves and
      add entry to available code list
      mark table entry as empty      // recycle their codes
bits_in_oldest++                     // update oldest leaves
IF bits_in_oldest > MAXBITS
   bits_in_oldest <- MINBITS         // wraparound

PROCEDURE outstring(code)
IF (code is a literal character)
   write code to output buffer
   RETURN
WHILE (code is not a literal character)
   push code.char                 // push the character for this node
   code<-code.basecode               // jump to the previous location
push code                          // save the last code (the literal)
pop string to output buffer          // pop the string we've built
```

Expanding LZW

Like the dynamic Huffman algorithm described earlier, LZW coding does not require you to pass a decoding table to the expander along with the compressed data. The LZW expander can build its own table from nothing but the codes in the compressed data.

The expansion program starts with a table, just like the compressor's, with only literal data defined. It begins by reading the first character from the compressed input. It sends this character to the output, but otherwise it just holds onto the character to form the basis for the next string.

For each code after the first that the expander reads, it generates a string and makes an update to the string table. The expander first uses the string table to translate the code value to an output string. For nonliteral codes, it backtracks through the code/character combinations of the string table, pushing characters onto a stack as it goes. When the expander reaches a literal code, it pops the stack to produce the output string.

In addition, each code after the first one causes a table update. For the second code, the expander adds a code made up of the first code, plus the first character in the string described by the second code. For each code thereafter, the expander adds the last code translated plus the first character in the current string to the table. The resulting table is an exact duplicate of the compression table, which changes with each code received.

Welch describes a special-case situation that complicates the expansion algorithm slightly. A certain type of string can cause the compressor to output a code before the expander has it in its table. This situation occurs when strings of the form *XandXandX* appear and the string *Xand* is already in the table. In this case, the compressor will send the code for *Xand* (because it already knows that string) and then add *XandX* to the table. It will then start with the middle *X*, find the next group of characters that it knows is *XandX*, and send the code for *XandX* before the expander knows its meaning.

You can handle this special case by adding a few lines of code in the expander program. If the expander receives a code that it doesn't recognize, it knows that it has encountered this singular case. In the above example, the expander receives the code for *Xand* and then an unknown code. It writes out the last translated code again (*Xand*) and then the first character from that code (*X*). It then adds a combination of these characters (*XandX*) to the table, which puts it back in sync with the compressor.

Enhanced LZW

Two enhancements to the basic LZW algorithm, variable-length codes and table clearing, make for a more flexible and robust compressor.

With fixed-length output codes, you must decide up front how many bits to use for encoding the compressed data. If you use a small number of bits, the table fills quickly and compression drops off rapidly. If you use a large number of bits, the overhead for each code that you do not successfully compress is enormous.

The sample code (see listing 2) uses variable-length codes to work around this problem. Initially, it uses 9-bit codes. When the compressor runs out of 9-bit codes, it switches to 10 bits, and on up through 13. It then uses 13-bit codes for the rest of the output.

Listing 2 shows that the compressor increases the bit length when the next code to add to the table requires more bits than allowed by the current bit length. This is not the next code to output; the next code to output will be the code that matches the next part of the input. However, because the expander and compressor use this same method for determining which code in the table to use next, they make the switch in bit sizes simultaneously.

Even with a 13-bit table, the LZW compressor will eventually run out of string locations. One way to handle this problem is to stop adding entries and use the strings in the table to compress the rest of the input. This will result in poor compression if the type of data changes from one part of the input to another.

You could also clear the table when it becomes full and start building the table again with the new data. Although this method makes the compressor more flexible than the do-nothing approach, it will also result in reduced compression while the table is mostly empty.

Listing 2 uses a partial-clearing approach to freshen the string table when it becomes full. The code clears only some of the older strings in the table when it becomes necessary.

Because the string data is stored as base code/character combinations, you can't merely keep track of the least frequently or least recently used strings in the table and later eliminate them when the table is full. You can eliminate only the nodes that are not used by other codes as base codes (the leaves).

It makes sense to keep track of the age of each leaf by the number of bits required to describe its code. When the table fills, you can remove all the 9-bit leaves and reuse their codes. When the table fills again, you can recycle the 10-bit leaf codes. Once the 13-bit leaves have been reused, you can go back to removing 9-bit leaves and continue in this manner indefinitely.

To determine which node is a leaf and which is not, the table-clearing routine takes a relatively brute force approach. First, it marks all the nodes as leaves. It then goes through the table, looking at base codes. Each base code is the code of a node that is not a leaf, so it unmarks the node that corresponds to that code.

Unfortunately, there is one more complication in finding the leaves to eliminate. The compressor stores its string table in a hashed array because it must try to find codes in the table knowing only their base codes and appended characters. For the clearing routine to find codes given the code itself, you can use a cross-index table that maps sorted codes to table locations. While this uses up a good chunk of memory (16K bytes for 16-bit pointers and a 13-bit table), it provides for quick table access by either the code or contents.

The Sample Code

To try out these two compression algorithms, I wrote two assembly routines designed to be called from programs written in C. (The full text of these routines is available in electronic format. See page 5 for details.) Both take an input and output file handle, compressing data from the input file and writing it to the output file.

Even if you don't need to write your own compressor, a little background in data compression is useful. Although data compression may appear complex and fraught with danger, it's actually valuable and reliable, as you can see. ■

REFERENCES

1. Storer, James A. *Data Compression: Methods and Theory*. Computer Science Press, 1988.
2. Vitter, J. S. "Design and Analysis of Dynamic Huffman Codes." *Journal of the Association for Computing Machinery*, October 1987.
3. Welch, Terry A. "A Technique for High Performance Data Compression." *Computer*, June 1984.

Steve Apiki is a BYTE Lab testing editor/engineer. You can reach him on BIX as "apiki."

Your questions and comments are welcome. Write to: Editor, BYTE, One Phoenix Mill Lane, Peterborough, NH 03458.